

Lecture on the Accumulation Buffer: Motion Blur, Anti-Aliasing, and Depth of Field

1 The Accumulation Buffer

There are a number of effects that can be achieved if you can draw a scene more than once. You can do this by using the *accumulation buffer*.

You can request an accumulation buffer with

```
glutInitDisplayMode(... | GLUT_ACCUM);
```

Conceptually, we're doing the following:

$$I = f_1 \boxed{\text{frame 1}} + f_2 \boxed{\text{frame 1}} + \dots + f_n \boxed{\text{frame n}}$$

We initialize the accumulation buffer to zero, then add in each frame, with an associated constant, and then copy the result to the frame buffer. This is accomplished by

- Clear the accumulation buffer with

```
glClearAccum(r, g, b, a);  
glClear(GL_ACCUM_BUFFER_BIT);
```

This is just like clearing the color buffer or the depth buffer. The accumulation buffer is like a running sum, though, so you will usually initialize it to zero.

- Add a frame into the accumulation buffer with:

```
glAccum(GL_ACCUM, f_i)
```

This function allows other values (see the man page), but we'll use this today.

- Copy the accumulation buffer to the frame buffer:

```
glAccum(GL_RETURN, 1.0);
```

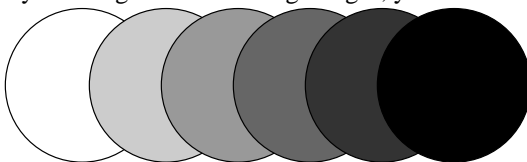
The second argument is a multiplicative factor for the whole buffer. We'll always use 1.0.

1.1 Advice

- Get your scene working first, before adding the accumulation buffer stuff.
- Make sure you clear the accumulation buffer to zero.
- Make sure your factors add to 1.

2 Motion Blur

By drawing a trail of fading images, you can simulate the blur that occurs with moving objects:



You can fade the images by using smaller coefficients on those images.

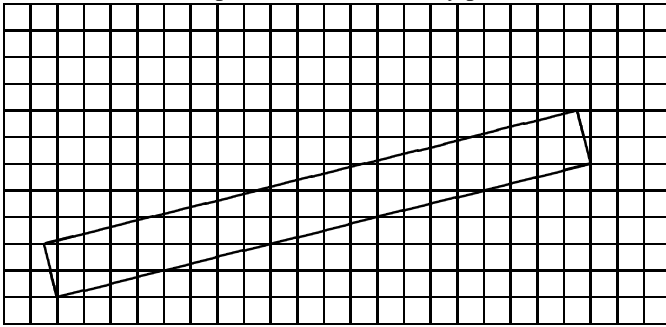
Demo: `~cs307/public_html/demos/accumulation/FallingTeapot.cc`

- Notice the three images of the teapot. Even without the accumulation buffer, just by drawing it three times, we can get an interesting effect.
- Type 's' and notice how if we draw them with a small motion, the teapot just looks bigger or distorted. Type 's' to return to the large motion.
- Type 'm' to switch to using the accumulation buffer. Notice how the frames can be made to vary in "strength." In fact, the first frame has more-or-less disappeared!
- Type 's' to use small motion and see the result. The teapot looks blurred by motion, which is roughly the effect we want. (Though if you think the effect without the accumulation buffer is nicer, I wouldn't blame you.)
- Type '+' or '-' to increase/decrease the number of frames.
- Looking at the code, notice how the accumulation buffer is initialized, drawn into, and read out of.
- Notice how the motion is interpolated based on the frame number. This is based on our parametric equations!
- Notice how the fractions for each frame are computed. What is the fraction for the frame that disappeared?

The problem with using the accumulation buffer for large-motion motion blur is that we really want to turn all of the coefficients up, so that the first image doesn't fade away and the last isn't too pale, yet that's mathematically nonsense and also doesn't work (the table turns *blue!*). Nevertheless, for small-motion blur, it works pretty well.

3 Anti-Aliasing

An important use of the accumulation buffer is in *anti-aliasing*. Aliasing is the technical term for "jaggies." It comes about because of the imposition of an arbitrary pixelation over a real world.



The idea of anti-aliasing using the accumulation buffer is:

- The scene gets drawn multiple times with slight perturbations (jittering), so that
- Each pixel is a local average of the images that intersect it.

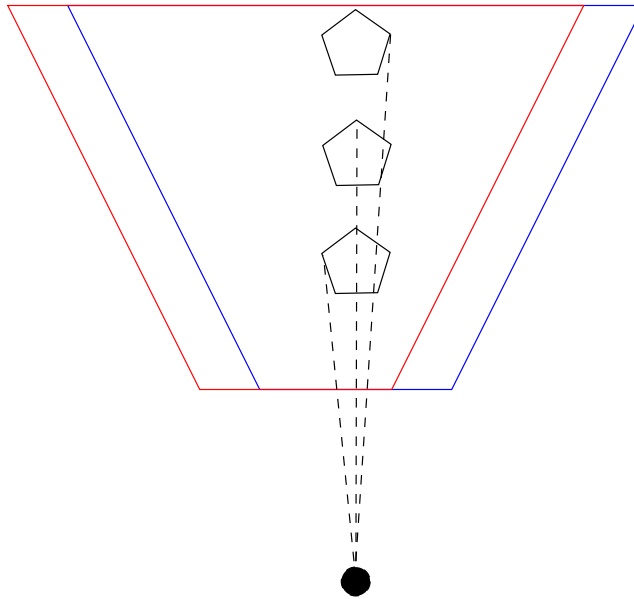
Generally speaking, you have to jitter by less than one pixel.

Demos: `~cs307/public_html/demos/accumulation/Aliasing.cc`

- Without anti-aliasing turned on, look at the four figures (callbacks 1-4) and notice the jaggies at the edges.
- Turn on anti-aliasing (callback 'a'), and notice the difference.
- Notice also that it doesn't quite work for the wire cube. Why not? Because the jitter amount is too small at the back and too big at the front.

4 Better Anti-Aliasing

A better technique than jittering the objects is to jitter the camera, or more precisely, to modify the frustum just a little so that the pixels that images fall on are just slightly different. Again, less than one pixel.



Here's a figure that may help:

The red and blue cameras differ only by an adjustment to the location of the frustum. The center of projection (the big black dot) hasn't changed, so all the rays still project to that point. The projection rays intersect the two frustums at different pixel values, though, so by averaging these images, we can anti-alias these projections.

How much should the two frustums differ, though? By less than one pixel. How can we move them by that amount? We only have control over *left*, *right*, *top* and *bottom*, and these are measured in world coordinates, not pixels. We need a conversion factor.

We can find a conversion factor in a simple way: the width of the frustum in pixels is just the width of the window (more precisely, the viewport), while the width of the frustum in world coordinates is just *right* - *left*. Therefore, the adjustment is:

$$\Delta x_{units} = \Delta x_{pixels} \frac{right - left}{windowwidth}$$

Here's the code, adapted from the OpenGL Programming Guide:

```
void accCamera(GLfloat pixdx, GLfloat pixdy) {
    GLfloat dx, dy;
    GLint viewport[4];
    glGetIntegerv(GL_VIEWPORT, viewport);

    GLfloat windowHeight=viewport[2];
    GLfloat windowHeight=viewport[3];
    GLfloat frustumWidth=right-left;
    GLfloat frustumHeight=top-bottom;

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    dx = pixdx*frustumWidth/windowWidth;
    dy = pixdy*frustumHeight/windowHeight;
    printf("world delta = %f %f\n",dx,dy);
    glFrustum(left+dx,right+dx,bottom+dy,top+dy,near,far);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

```
}
```

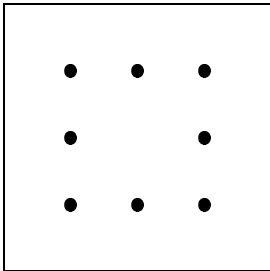
- The pixdx and pixdy values are the jitter amounts (distances) in pixels (sub-pixels, actually).
- The frustum is altered in world coordinates
- Therefore, dx and dy are computed in world coordinates corresponding to the desired distance in pixels.

```
void smoothDisplay() {  
    int jitter;  
    int numJitters = 8;  
    glClear(GL_ACCUM_BUFFER_BIT);  
    for(jitter=0; jitter<numJitters; jitter++) {  
        accCamera(jitterTable[jitter][0],  
                jitterTable[jitter][1]);  
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
        drawObject();  
        glAccum(GL_ACCUM, 1.0/numJitters);  
    }  
    glAccum(GL_RETURN, 1.0);  
    glFlush();  
    glutSwapBuffers();  
}
```

This does just what you think:

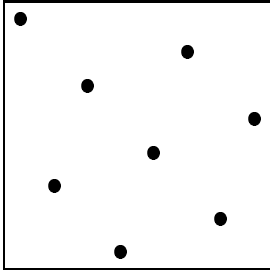
- We draw the image 8 times, each time adjusting the pixel jitter.
- Each drawing has equal weight of 1/8.

There are lots of ways to imagine how the pixel jitter distances are computed. The domino pattern is a good idea for 8.



However, a paper on the subject argues for the following, which I just used without further investigation. One idea I have is that we want to avoid regular patterns.

```
// From the OpenGL Programming Guide, first edition, table 10-5  
float jitterTable[][2] = {  
    {0.5625, 0.4375},  
    {0.0625, 0.9375},  
    {0.3125, 0.6875},  
    {0.6875, 0.8124},  
    {0.8125, 0.1875},  
    {0.9375, 0.5625},  
    {0.4375, 0.0625},  
    {0.1875, 0.3125}};
```



Demos: `~cs307/public_html/demos/accumulation/AntiAliasing.cc`

- Notice the difference in quality between the two images
- We'll look at how the code is written, but it follows what we've shown above.

This better approach to anti-aliasing works regardless of how far the object is from the center of projection, unlike the object-jitter we did before. Furthermore, we have a well-founded procedure for choosing the jitter amount, not just trial and error.

5 Depth of Field

Another thing we can do with the accumulation buffer is to blur things that a camera would blur. We'll investigate

- why cameras blur things
- what is meant by "focal depth"
- what is meant by "depth of field"
- how to accomplish this by using the accumulation buffer

Here's a very good web site in terms of the figures and the pictures of the rulers. I've also added this link to our course home page.

<http://www.cs.mtu.edu/~shene/DigiCam/User-Guide/950/depth-of-field.html>

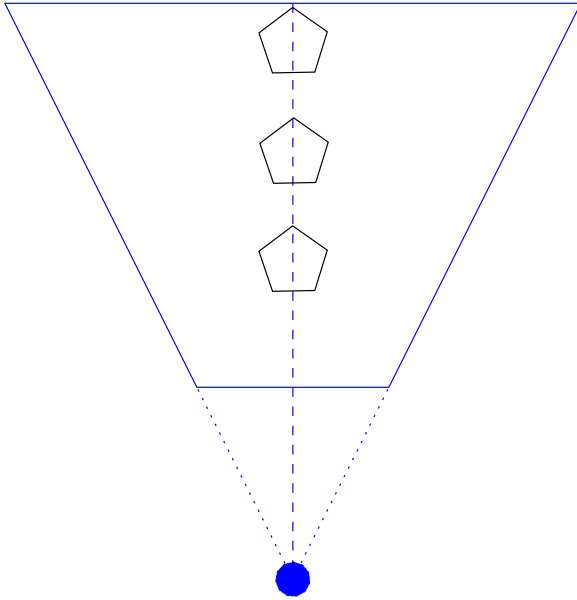
Ignore what it says about the circles of confusion being caused by the amount of light going through the lens. The circles of confusion are caused by the *aperture size* (which also controls the amount of light, but the causality goes the other way):

Smaller aperture means:

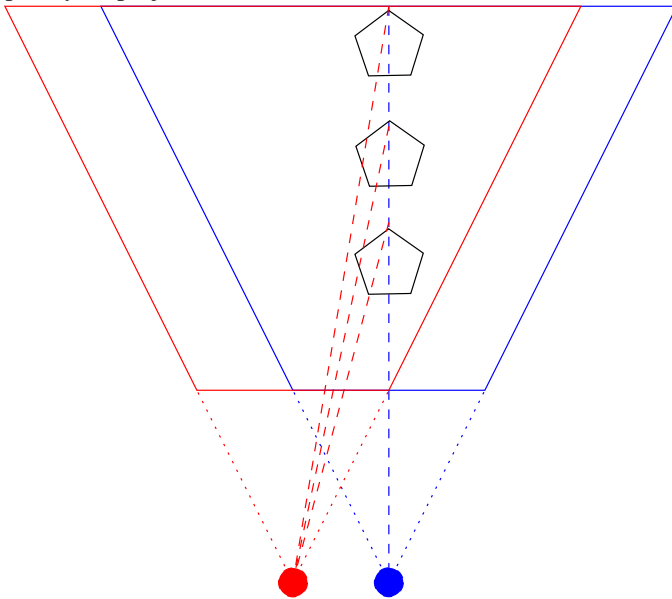
- greater depth of field, and
- less light

Of course, in CG, we don't have apertures. Instead, we have to fake the blurriness. But how do we blur the things except at the focal distance?

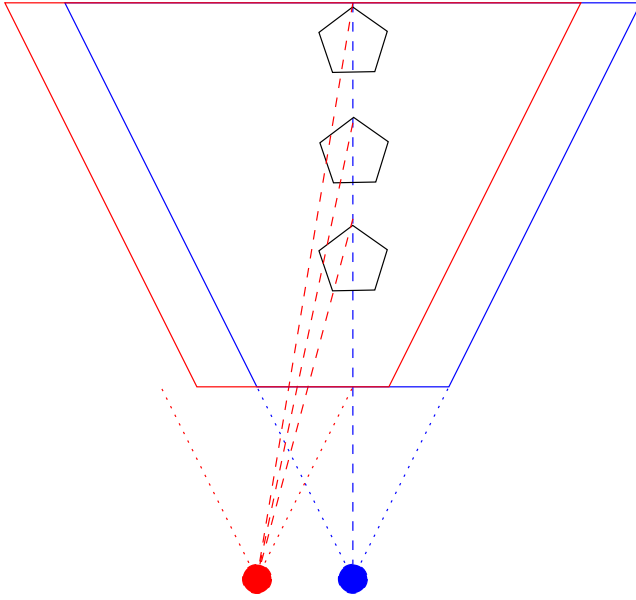
First, let's remind ourselves of the basic camera terminology, since our figures are about to get very complicated. The big blue dot is the eye location; everything about the camera is measured from that point. In particular, *left*, *right*, *top* and *bottom* are measured with respect to that. The frustum is the blue trapezoid. Make sure you understand what left, right, top and bottom are. The dashed line is where some vertices project; in this case they happen to project to the center of the frustum, but that's just to simplify some of the geometry we'll do later.



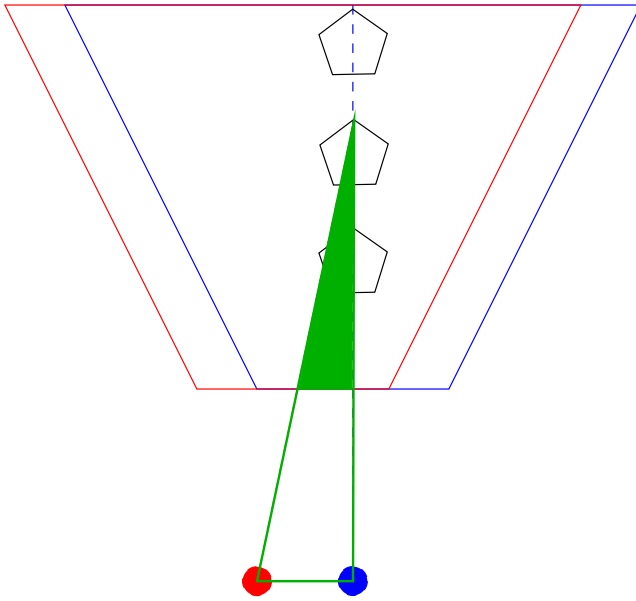
Next, notice that if we move the camera, points project onto different pixels. In the following figure, the difference is that we've moved the eye to the left (red) location; the camera moves as a result of moving the eye. The dashed lines show where the three vertices project to. In the blue setup, they all project to the center of the frustum. In the red setup, they all project to different locations.



That doesn't accomplish our depth-of-field goal, of course, because all the figures (the pentagons) are blurred. Suppose we want to make the middle one non-blurred. What we do is *adjust* the frustum's location, as it's measured from the new eye location, to *cancel* out the effect of the eye movement, so that that one ray still projects to the center of the frustum.



The geometry is just by similar triangles:



If F is the focus distance, measured from the eye to the object (the height of the larger triangle) and Δx_e is the amount that we move the eye (the width of the larger triangle), we can see that:

$$\frac{\Delta x_e}{F} = \frac{\Delta x_f}{F - \text{near}}$$

So, we adjust the frustum by Δx_f . Note that to accomplish the desired cancelation, the two deltas have to have opposite signs: if we move the eye to the left (negative), we have to move the frustum to the right (positive). We find:

$$\Delta x_f = (\Delta x_e / F)(F - \text{near})$$

The code for setting up our camera (frustum), then, is:

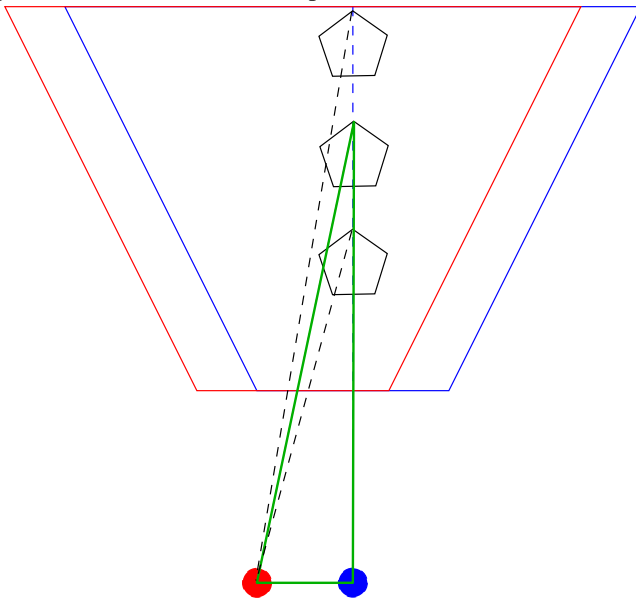
```
/* Set up camera. Note that near, far, and focus remain fixed; all that
is modified with the accumulation buffer is the eye location.*/
void accCamera(GLfloat eyedx, GLfloat eyedy, GLfloat focus) {
    GLfloat dx,dy;
```

```

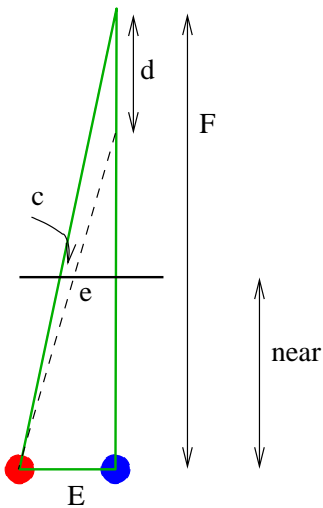
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
dx = (eyedx/focus)*(focus-near);
dy = (eyedy/focus)*(focus-near);
printf("dx = %.5f, dy = %.5f\n", dx, dy);
glFrustum(left+dx,right+dx,bottom+dy,top+dy,near,far);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
// Translating the coordinate system by (eyedx,eyedy) is the same as
// moving the eye by (-eyedx,-eyedy). Nobody better re-load the
// identity matrix!
glTranslatef(eyedx,eyedy,0.0);
}

```

The next question is how far to move the eye. The farther the eye moves, the narrower the depth of field. First, look at this figure. The blurriness of the other two figures depends on the difference between where they project to in the two camera setups.



To understand this more, I want to focus on two pairs of similar triangles:



One pair gives us the following:

$$\frac{F}{E} = \frac{F - \text{near}}{e}$$

The other pair gives us

$$\frac{F - d}{E} = \frac{F - d - \text{near}}{e - c}$$

What are these values?

- near: the distance from the eye to the frustum
- F: the distance where objects are in focus
- E: the (maximum) distance the eye moves
- d: the desired depth of field
- c: the blurriness distance, about 1 pixel

The unknown is E . We'd like to compute E as a function of d . By eliminating e between the two equations, we can do this. Skipping some horrendous algebra (check me on this, please), we get

$$E = \frac{-c}{\frac{F-d-\text{near}}{F-d} - \frac{F-\text{near}}{F}}$$

Note that all these values are in world units except for c , which is about 1 pixel. So, we need to convert c from pixels to world units before we do this computation.

Demos: `~cs307/public_html/demos/accumulation/DepthOfField.cc`